类型系统深入

类型系统深入
- 学习目标
- 类型保护
  - typeof
  - instanceof
  - in
  - 字面量类型保护
  - 自定义类型保护
- 类型操作
  - typeof
  - keyof
  - in
- 类型兼容

学习目标

- 理解什么是类型保护，并能使用类型保护去优化代码逻辑
- 熟悉类型操作符的使用，进一步理解类型系统

类型保护

我们通常在 `JavaScript` 中通过判断来处理一些逻辑，在 `TypeScript` 中这种条件语句块还有另外一个特性：根据判断逻辑的结果，缩小类型范围（有点类似断言），这种特性称为 `类型保护`，触发条件：

- 逻辑条件语句块：if、else、elseif
- 特定的一些关键字：typeof、instanceof、in.....

typeof

我们知道 `typeof` 可以返回某个数据的类型，在 `TypeScript` 在 `if`、`else` 代码块中能够把 `typeof` 识别为类型保护，推断出适合的类型

```
function fn(a: string|number) {
  // error, 不能保证 a 就是字符串
  a.substring(1);
  if (typeof a === 'string') {
    // ok
    a.substring(1);
  } else {
    // ok
    a.toFixed(1);
  }
}
```

instanceof

与 `typeof` 类似的，`instanceof` 也可以被 `TypeScript` 识别为类型保护

```
function fn(a: Date|Array<any>) {
  if (a instanceof Array) {
    a.push(1);
  } else {
    a.getFullYear();
  }
}
```

in

`in` 也是如此

```
interface IA {
    x: string;
    y: string;
}
interface IB {
    a: string;
    b: string;
}

function fn(arg: IA | IB) {
    if ('x' in arg) {
        // ok
        arg.x;
        // error
        arg.a;
    } else {
        // ok
        arg.a;
        // error
```

```
        arg.x;
    }
}
```

字面量类型保护

如果类型为字面量类型，那么还可以通过该字面量类型的字面值进行推断

```
interface IA {
    type: 'IA';
    x: string;
    y: string;
}
interface IB {
    type: 'IB';
    a: string;
    b: string;
}

function fn(arg: IA | IB) {
    if (arg.type === 'IA') {
        // ok
        arg.x;
        // error
        arg.a;
    } else {
        // ok
        arg.a;
        // error
        arg.x;
    }
}
```

自定义类型保护

有的时候，以上的一些方式并不能满足一些特殊情况，则可以自定义类型保护规则

```
function canEach(data: any): data is Element[]|NodeList {
    return data.forEach !== undefined;
}
function fn2(elements: Element[]|NodeList|Element) {
    if ( canEach(elements) ) {
        elements.forEach((el: Element)=>{
            el.classList.add('box');
        });
    } else {
        elements.classList.add('box');
    }
}
```

`data is Element[]|NodeList` 是一种类型谓词，格式为：`xx is XX` ，返回这种类型的函数就可以被 `TypeScript` 识别为类型保护

类型操作

`TypeScript` 提供了一些方式来操作类型这种数据，但是需要注意的是，类型数据只能作为类型来使用，而不能作为程序中的数据，这是两种不同的数据，一个用在编译检测阶段，一个用于程序执行阶段

typeof

在 `TypeScript` 中，`typeof` 有两种作用

- 获取数据的类型
- 捕获数据的类型

```
let str1 = 'kaikeba';

// 如果是 let , 把 'string' 作为值
let t = typeof str1;

// 如果是 type, 把 'string' 作为类型
type myType = typeof str1;

let str2: myType = '开课吧';
let str3: typeof str1 = '开课吧';
```

keyof

获取类型的所有 `key` 的集合

```
interface Person {
    name: string;
    age: number;
```

```
};
type personKeys = keyof Person;
// 等同：type personKeys = "name" | "age"

let p1 = {
    name: 'zMouse',
    age: 35
}
function getPersonVal(k: personKeys) {
    return p1[k];
}
/**
等同：
function getPersonVal(k: 'name'|'age') {
    return p1[k];
}
*/
getPersonVal('name'); //正确
getPersonVal('gender'); //错误
```

in

针对类型进行操作的话，内部使用的 `for…in` 对类型进行遍历

```
interface Person {
    name: string;
    age: number;
}
type personKeys = keyof Person;
type newPerson = {
    [k in personKeys]: number;
    /**
    等同 [k in 'name'|'age']: number;
    也可以写成
    [k in keyof Person]: number;
    */
}
/**
type newPerson = {
    name: number;
    age: number;
}
*/
```

注意：`in` 后面的类型值必须是 `string` 或者 `number` 或者 `symbol`

类型兼容

`TypeScript`的类型系统是基于结构子类型的，它与名义类型（如：java）不同（名义类型的数据类型兼容性或等价性是通过明确的声明或类型的名称来决定的）。这种基于结构子类型的类型系统是基于组成结构的，只要具有相同类型的成员，则两种类型即为兼容的。

```
class Person {
    name: string;
    age: number;
}
class Cat {
    name: string;
    age: number;
}

function fn(p: Person) {
    p.name;
}

let xiaohua = new Cat();
// ok，因为 Cat 类型的结构与 Person 类型的结构相似，所以它们是兼容的
fn(xiaohua);
```